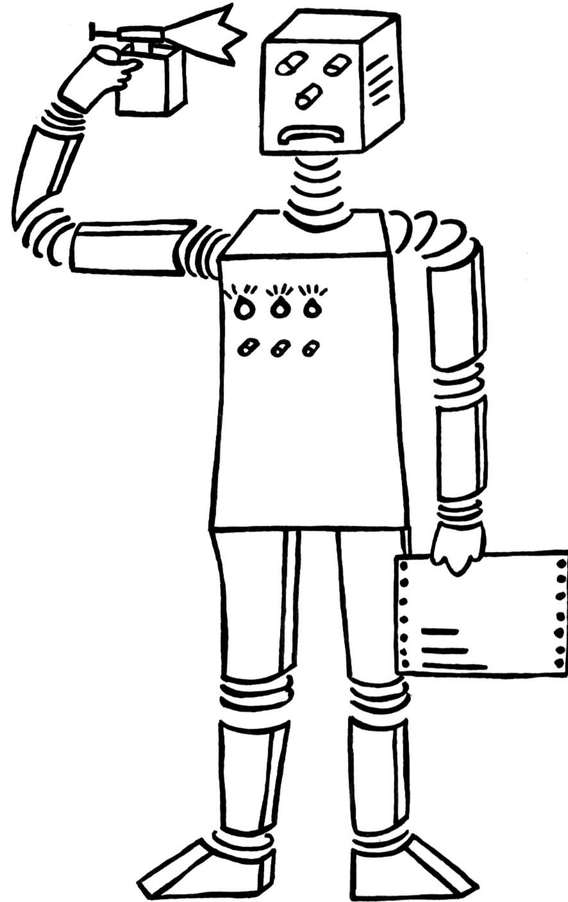
Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany

SEKI-Working Paper



Iterativer LISPLOG Interpreter

Implementierung, Dokumentation und Evaluation

Michael Dahmen

Juni 1986 SWP-86-03

ITERATIVER LISPLOG INTERPRETER

Implementierung, Dokumentation und Evaluation

Michael Dahmen

Fachbereich Informatik
SFB 314 Bau 14
Universitaet Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1
W. Germany

uucp: unido!uklirb!dahmen
oder dahmen@uklirb.UUCP

SEKI WORKING PAPER
SWP-86-03

Juni 1986

Was das Optimieren betrifft, befolgen wir zwei Regeln :

Regel 1 : Tu's nicht

Regel 2 : (nur fuer Experten)
Tu's nicht - d.h. nicht bevor Du eine vollkommen klare und nichtoptimierte Loesung hast

M.A. Jackson

Zusammenfassung :

Diese Arbeit beschreibt den iterativen Interpreter LISPLOG.2, der aus einer vorherigen, rekursiven Version (LISPLOG.1) entwickelt wurde. Das Ziel dieser Weiterentwicklung war eine Leistungssteigerung des Interpreters, insbesondere eine Beschleunigung des Programmablaufs. Im letzten Abschnitt dieser Arbeit wird anhand von Laufzeitvergleichen gezeigt, in wieweit dieses Ziel erreicht werden konnte.

Im ersten Teil dieser Arbeit wird aufgezeigt, worin sich die Interpreterversionen unterscheiden und worauf die Leistungsteigerung beruht. Ausserdem wird auf weitere Ansatzpunkte fuer noch schnellere Implementierungen eingegangen.

Teil zwei dokumentiert das ebenfalls in dieser Arbeit vorliegende Programmlisting des LISPLOG.2-Interpreters. Da diese Arbeit auf den vorherigen ueber LISPLOG.1 aufbaut, ist es zum Verstaendnis manchmal notwendig, auf die Documentation des rekursiven Interpreters zurueckzugreifen.

In den Anhaengen findet man neben dem Listing des LISPLOG.2-Interpreters auch die zur Messung von Laufzeit und Speicherverbrauch erstellten Programme.

Diese Arbeit ist im Sonderforschungsbereich 314, "Kuenstliche Intelligenz - Wissensbasierte Systeme", in Kaiserslautern entstanden.

Abstract :

This paper discusses the iterative interpreter LISPLOG.2, which was developed from the earlier recursive version LISPLOG.1. The goal of this development was to speed up the execution of LISPLOG programs. A comparison between the two LISPLOG versions - given in chapter three - shows to what extent this goal could be achieved.

The first chapter of this paper describes the differences between the two versions and shows how these differences effect the performance.

Chapter two is the documentation of the main aspects of LISPLOG.2. Since LISPLOG.2 was developed from LISPLOG.1 it may be useful to consult the documentation of LISPLOG.1 too.

The appendices include the listing of the LISPLOG.2 interpreter and the programs used for the performance analysis.

This research was supported by the
Deutsche Forschungsgemeinschaft, Sonderforschungsbereich 314, "Kuenstliche Intelligenz - Wissensbasierte Systeme".

Inhalt :

1. Funktionsprinzipien der LISPLOG-Interpreter

1.1. Merkmale und Eigenschaften von LISPLOG.1

1.2. Arbeitsweise von LISPLOG.2
1.2.1. Iteration statt Rekursion
1.2.2. Zugriff auf Variablenbindungen
1.2.3. Umbenennen von Klauseln
1.2.4. Zusammenfassen von Aktionen
1.2.5. Unifikation

1.3. Weitere Optimierungsmoeglichkeiten
1.3.1. Structure Sharing
1.3.2. Indexieren der Variablen

2. Dokumentation zu LISPLOG.2

2.1. Ablauf eines LISPLOG.2-Beweises
2.1.1. Argumente der Funktion "prove"
2.1.2. Bedeutung der lokalen Variablen von "prove"
2.1.3. Aufbau der Stacks
2.1.4. Kontrollfluss in der Funktion "prove"

2.2. Speicherung des LISPLOG.2-Environment
2.2.1. Datenstrukturen
2.2.2. Operationen
2.2.3. Schematische Darstellung von Einfuegen und Loeschen
2.2.4. Variablen in LISP-Aufrufen von LISPLOG

3. Laufzeitvergleiche

3.1. Vorbemerkungen

3.2. Verwendete Testprogramme

3.3. Masseinheit und Messverfahren

3.4. Messergebnisse

3.5. Anmerkungen

4. Literatur

Anhang 1 : Progammlistings des LISPLOG.2-Interpreters

Anhang 2 : Laufzeitanalyse von LISP-Programmen

Anhang 3 : Speicherverbrauchsanalyse fuer LISP-Programme

1. Funktionsprinzipien der LISPLOG-Interpreter

Der LISPLOG.2-Interpreter wurde aus dem in [Boley & Kammermeier et al. 1985] beschriebenen Interpeter LISPLOG.1 entwickelt. Obwohl dabei eine erhebliche Leistungssteigerung erzielt wurde, konnten grosse Teile des LISPLOG.1-Interpreters fast ungeaendert uebernommen werden. Die folgenden Betrachtungen beziehen sich daher nur auf die Teile, an denen wesentliche Aenderungen vorgenommen wurden.

1.1. Merkmale und Eigenschaften von LISPLOG.1

Die wesentlichen Merkmale des LISPLOG.1-Interpreters, im Bezug auf das hier diskutierte Thema, sind :

- Korekursive Kontrollstruktur
  Der LISPLOG.1-Interpreter benutzt zwei Hauptfunktionen ("and-process", "or-process"), die sich solange gegenseitig aufrufen, bis die Anfrage bewiesen ist. Daraus resultiert ein stark funktionaler Programmstil und ein hoher LISP-Stack-Bedarf.

- Environment gespeichert als Liste
  Die Bindungsumgebung wird als lineare Liste gespeichert. Das ist die einfachste aber auch die langsamste Organisation.

- Aufwendiges Umbenennen
  Die Klauseln werden vor einer Unifikation vollstaendig umbenannt. Diese Operation erfordert sehr viel Zeit und Speicherplatz. Das ist oft unnoetig, und kann durch Verzoegerung dieser Operation auf einen spaeteren Zeitpunkt erheblich beschleunigt werden.

Aus diesen Eigenschaften resultieren zwei schwerwiegende Probleme bei der Benutzung des Interpreters :

- eingeschraenkte Beweislaenge
  Es koennen nur relativ kurze Beweise durchgefuehrt werden, da der Stackbedarf (Belastung des LISP-Laufzeitstacks) sehr hoch ist und der LISP-Stack nicht ohne weiteres vergroessert werden kann. Dies liegt im wesentlichen an der korekursiven Struktur ("and-or-process") des Interpreters.

- geringe Geschwindigkeit
  Die Beweise nehmen erhebliche Zeit in Anspruch, wesentlich laenger als bespielsweise aequivalente CPROLOG-Programme (siehe Abschnitt 3). Die Ursache dafuer ist zum einen, dass der LISPLOG.1-Interpreter in LISP implementiert ist, der CPROLOG-Interpreter dagegen in C. Ausserdem ist der LISPLOG.1-Interpreter, wie bereits erwaehnt, nicht besonders effizient realisiert.

Bevor man eine Optimierung des Laufzeitverhaltens vornehmen kann, sollte man analysieren, welche Funktionen im wesentlichen die Laufzeit bestimmen. Zu diesem Zweck wurde ein Messprogramm entwickelt, dass die Bestimmung des Laufzeitbedarfs einzelner Funktionen aus einem Programmpaket erlaubt (siehe Anhang 2). Als Testanwendung wurde ein LISPLOG-Programm (Schachendspiel; siehe auch Abschnitt 3) benutzt. Eine Messung der Laufzeit, aufgeschluesselt nach Funktionsgruppen, ergab folgendes Ergebnis fuer den LISPLOG.1-Interpreter :

Umbenennen der Klauseln	: 40 %	(180 %)
Unifikation	: 25 %	(112,5 %)
Zugriff auf Variablenbindungen	: 25 %	(112,5 %)
Steuerung (and-or-process)	: 10 %	(45 %)
reine Rechenzeit	: 100 %	(450 %)

In Klammern Werte im Vergleich zur Laufzeit von LISPLOG.2 (100% = gesamte Laufzeit fuer LISPLOG.2; siehe auch Abschnitt 3.4).

Neben der reinen Rechenzeit ist noch Zeit fuer die Speicherbereinigung (Garbage Collection) erforderlich. Die Zeit dafuer haengt im wesentlichen von der Groesse des Speichers, dem verwendeten Algorithmus zur Speicherbereinigung und dem Speicherbedarf des Interpreters ab. Diese Faktoren koennen aber durch Aenderungen im Interpreter nur zum Teil beeinflusst werden.

reine Rechenzeit	: 90 %	(450 %)
Garbage Collection	: 10 %	(50 %)
gesamte Laufzeit	: 100 %	(500 %)

1.2. Arbeitsweise von LISPLOG.2

Gegenueber dem LISPLOG.1-Interpreter wurden die folgenden Aenderungen vorgenommen.

1.2.1. Iteration statt Rekursion

Der LISPLOG.2-Interpreter verwaltet die noch unbewiesenen Teilziele und die noch offenen Backtrackmoeglichkeiten in zwei eigenen Stacks. Dadurch wird der LISP-Name-Stack wesentlich weniger belastet. Der LISPLOG.2-Interpreter kann daher wesentlich laengere Beweise durchfuehren als der LISPLOG.1-Interpreter.

1.2.2. Zugriff auf Variablenbindungen

Der LISPLOG.2-Interpreter benutzt statt einer linearen Liste eine Array-Struktur, bei der sowohl der Variablenname als auch die Ebene (Level) der Variable als Index verwendet wird. Dadurch wird die Zugriffszeit nahezu unabhaengig von der Anzahl der gespeicherten Bindungen. Der Preis fuer diese Effizienzsteigerung ist eine explizite Verwaltung der Bindungen, d.h. ungueltige Bindungen muessen explizit geloescht werden (bei LISPLOG.1 automatisch durch LISP-Stackverwaltung).

1.2.3. Umbenennen von Klauseln

Im LISPLOG.1-Interpreter werden die Variablen in den Klauseln umbenannt noch bevor feststeht, ob die Klausel ueberhaupt anwendbar ist. Im LISPLOG.2-Interpreter wird die Umbenennung verschoben bis sicher ist, dass sie erforderlich ist; d.h. Klauselkoepfe werden waehrend des Unifikationsschritts umbenannt, die Klauselruempfe werden erst umbenannt, wenn sie als Ziel benutzt werden. Durch diese Massnahmen wird der Aufwand fuer die Umbenennung deutlich reduziert.

1.2.4. Zusammenfassen von Aktionen

Die Aktionen Unifikation, Umbenennung und Instanziierung muessen jeweils einen Term vollstaendig bis auf die Ebene der Atome zerlegen. Fasst man diese Aktionen zusammen wo dies moeglich ist, so kann der Gesamtaufwand reduziert werden. Im LISPLOG.2-Interpreter wurden einmal Unifikation und Umbenennung sowie zum zweiten Umbenennung und Instanziierung zusammengefasst.

1.2.5. Unifikation

Am Unifkationsalgorithmus wurde nur eine Aenderung vorgenommen, die die Anzahl der Vergleiche reduziert (Ersetzung von "equal" durch "eq"). Obwohl diese Aenderung unbedeutend erscheint reduziert sie doch die Laufzeit deutlich, denn der Vergleich mit "equal" ist sehr aufwendig. Diese Einsparung ist moeglich, weil die Unifikation die beiden Argumente ohnehin vollstaendig in Atome und Variablen zerlegt. Es gibt allerdings auch einige seltene Faelle, in denen diese Aenderung zu einer Erhoehung der Laufzeit fuehrt (siehe Abschnitt 3.5).

Durch diese Massnahmen ist der LISPLOG.2-Interpreter mehr als 5-mal schneller als LISPLOG.1. Auch fuer LISPLOG.2 wurde die Laufzeit nach Funktionsgruppen aufgeschluesselt, wobei genauso vorgegangen wurde wie in 1.1 beschrieben.

Umbenennung-Instanziierung	:	55 %	(30 %)
Unifikation-Umbenennung	:	25 %	(14 %)
Steuerung	:	20 %	(11 %)
reine Rechenzeit	:	100 %	(55 %)

(100% = gesamte Laufzeit fuer LISPLOG.2)

reine Rechenzeit	:	55 %	(55 %)
Garbage Collection	:	45 %	(45 %)
gesamte Laufzeit	:	100 %	(100 %)

Aus den angegebenen Werten erkennt man, dass der LISPLOG.2-Interpreter fast die Haelfte der Laufzeit fuer die Speicherbereinigung verbraucht. Daher ist auch durch eine Reduzierung des Speicherverbrauchs eine deutliche Geschwindigkeitssteigerung moeglich. Zur Analyse des Speicherverbrauchs wurde ein Messprogramm entwickelt, das in Anhang 3 angegeben ist. Mit diesem Programm wurden fuer den LISPLOG.2-Interpreter folgende Werte ermittelt :

Kopieren der Strukturen (Klauselteile)	:	40 %
Verwaltung von Environment	:	30 %
Verwaltung der LISPLOG-Stacks	:	20 %
Verwaltung von Trail	:	10 %
Gesamter Speicherverbrauch	:	100 %

1.3. Weitere Optimierungsmoeglichkeiten

Die Leistungen des LISPLOG.2-Interpreters sind zwar deutlich besser als die von LISPLOG.1, fuer wirklich grosse Anwendungen reicht aber die Arbeitsgeschwindigkeit noch nicht aus. Die Implementation des Interpreters in LISP ist sicherlich das (fuer die Laufzeit) entscheidende Problem dieses Ansatzes, ein in Assembler oder "C" realisierter Interpreter waere sicher schneller. Da aus anderen Gruenden (Portabilitaet, Integration) LISP als Basissprache beibehalten werden soll, sind die Moeglichkeiten fuer weitere interpretative Leistungssteigerungen begrenzt. Einen nennenswerten Effekt kann man noch von zwei Massnahmen erwarten, die nachfolgend kurz skizziert werden.

1.3.1. Structure Sharing

Sowohl LISPLOG.1 als auch LISPLOG.2 arbeiten mit Structure-Copying, d.h. sie kopieren die Klauseln beim Umbenennen. Dies erfordert nicht nur, auch in LISPLOG.2, den groessten Teil der Laufzeit, sondern ist auch fuer etwa 40 % des Speicherverbrauchs verantwortlich (s. o.). Bei einem Anteil der Speicherbereinigung von 45% an der Gesamtlaufzeit ist auch aus diesem Grund beim Uebergang auf Structure Sharing eine Erhoehung der Arbeitsgeschwindigkeit zu erwarten.

1.3.2. Indexieren der Variablen

Die in Abschnitt 2.2 beschriebene Speicherung des Environments im LISPLOG.2-Interpreter kann noch verbessert werden, wenn man schon beim Einlesen der Klauseln alle Variablennamen durch Indices ersetzt (vgl. [Herr 86]). Dies beschleunigt zwar die Ausfuehrung nicht besonders, vereinfacht aber die doch recht komplizierte Speicherung des Environments erheblich, wodurch ebenfalls Speicher eingespart werden kann.

Welcher Effekt durch diese beiden Massnahmen erreicht werden kann, ist nur schwer voraussagbar. Aufgrund der bisherigen Erfahrungen beim Uebergang von LISPLOG.1 auf LISPLOG.2 erscheint eine Geschwindigkeitssteigerung um einen Faktor 2 - 3 gegenueber LISPLOG.2 aber als moeglich.

Als Alternative zur skizzierten Weiterentwicklung des Interpreters ist auch die Entwicklung eines Compilers denkbar. Dieser Compiler wuerde LISPLOG-Programme in LISP-Programme uebersetzen, die dann vom LISP-Compiler in Maschinensprache uebersetzt wuerden. Darauf kann aber hier nicht weiter eingegangen werden.

2. Dokumentation zu LISPLOG.2

In diesem Abschnitt werden die wichtigsten Aenderungen gegenueber LISPLOG.1 beschrieben, wobei Bezug genommen wird auf das Programmlisting von LISPLOG.2 (Anhang 1).

2.1. Ablauf eines LISPLOG.2-Beweises

Die Hauptfunktion des Interpreters ist "prove". Diese Funktion arbeitet iterativ und ersetzt die korekursiven Funktionen "and-process" und "or-process" in LISPLOG.1. Dadurch ergeben sich erhebliche Abweichungen, die im folgenden beschrieben werden.

Die Funktion "prove" arbeitet mit zwei Stacks; der eine enthaelt die noch nicht bewiesenen Ziele ("goal-stack"), der andere die noch offenen Backtracking-Punkte ("choice-stack"). Der "goal-stack" wird durch Beweisschritte verkleinert und durch die Praemissen der Klauseln vergroessert. Der "choice-stack" waechst mit jedem Beweisschritt und wird verkleinert, wenn ein Backtracking durchgefuehrt wird. Der "goal-stack" entspricht in LISPLOG.1 dem Parameter "goal-list", der "choice-stack" wird in LISPLOG.1 durch die korekursiven Aufrufe realisiert.

In dem "choice-stack" wird der Zustand des Beweises festgehalten, wenn eine Klausel zur Resolution benutzt wird. Indem man ein Element von "choice-stack" entnimmt, kann man einen frueheren Zustand wiederherstellen und so ein Backtracking durchfuehren.

Der Zugriff auf LISP-Funktionen und LISPLOG-Primitive erfolgt genauso wie im LISPLOG.1-Interpreter mit den Funktionen "lisp-predicates" und "prolog-primitive".

2.1.1. Argumente der Funktion "prove"

Die Funktion "prove" hat drei Argumente.

1) Liste der zu beweisenden Ziele :
   Die zu beweisenden Ziele sind logisch Und-Verknuepft und in einer Liste zusammengefasst. Die Ziele koennen Variablen enthalten. Dabei ist zu beachten dass ein eventuell vorhandener Variablen-Level ignoriert wird, d.h. die Variablen haben die Form "(? <Name>)".

2) Anzahl der Loesungen :
   Ist dieses Argument eine positve Zahl n, so werden hoechstens n Loesungen berechnet und als Liste von "prove" zurueckgegeben. Ist dieses Argument keine Zahl, so wird der Benutzer nach jeder berechneten Loesung gefragt, ob er weitere Loesungen suchen lassen moechte.

3) Art der Loesung :
Dieses Argument ist nur von Bedeutung, wenn Argument 2 eine Zahl ist. In diesem Fall gibt "prove" eine Liste der Loesungen zurueck. Ist Argument 3 gleich "nil", so ist jede Loesung eine Liste der Bindungen aller Variablen, die in Argument 1 vorkammen. Ist das Argument 3 ein s-expression ungleich "nil", so wird dieser s-expression instanziiert, wenn der Beweis beendet ist. Jede Loesung ist dann ein instanziierter s-expression.

2.1.2. Bedeutung der lokalen Variablen von "prove"

Die Funktion "prove" benutzt und veraendert die folgenden Variablen. Mit Ausnahme von "environment" werden alle diese Variablen nur von "prove" beeinflusst.

"goal-stack" :
Diese Variable enthaelt die noch zu beweisenden Ziele. Jedes Ziel besteht aus dem LISPLOG-Literal, so wie es aus der Datenbasis entnommen wurde (Praemisse), und dem Level, auf das die Variablen in dem Literal umbenannt werden muessen. Daneben enthaelt der "goal-stack" auch Markierungen, die den Tracer zu einem spaeteren Zeitpunkt zur Ausgabe einer "Exit" Meldung veranlassen.

"choice-stack" :
Diese Variable enthaelt die noch offenen Backtracking Punkte des bislang durchgefuehrten Teilbeweises. Jeder Backtracking Punkt besteht aus einem 8-Tupel, das den Zustand des Beweises nach einer Unifikation festhaelt (s.u.).

"environment" :
Speichert alle im Verlauf des Beweises entstehenden Bindungen und wird nur durch die Operationen "env-add" und "env-rem" direkt beeinflusst.

"solutions" :
Sammelt alle berechneten Loesungen, wenn der Beweis nicht interaktiv sondern mit maximaler Zahl von Loesungen durchgefuehrt wird ("anzahl" ist eine Zahl). "solutions" ist eine Liste, deren Elemente entweder die Bindungen der Variablen auf Level 0 oder die Instanziierung des "result-term" Arguments von "prove" sind.

"goal" :
Das aktuell zu beweisende Ziel. Es wird vom "goal-stack" (bei "Call") bzw. vom "choice-stack" (beim Backtracking) entnommen.

"database-left" :
Enthaelt die noch verbleibenden Klauseln, die zur Resolution mit dem aktuellen Ziel in Frage kommen.

"clause" :
Die aktuell zur Resolution benutzte Klausel.

"trail" :
Eine Liste der Bindungen, die beim aktuellen Unifikationsschritt entstanden sind. Diese Liste wird benoetigt, um das globale "environment" zurueckzusetzten wenn ein Backtracking erfolgt. Die Operationen auf der Variable "environment" werden nachfolgend beschrieben.

"level-of-db" :
Enthaelt das Level, auf das die Klauseln in "database-left" noch umbenannt werden muessen.

"level-of-goal" :
Auf dieses Level wurden die Variablen im aktuellen "goal" umbenannt.

2.1.3. Aufbau der Stacks

Es gibt einen "goal-stack" und einen "choice-stack", die folgende Form haben :

```
<goal-stack>     = ( <Praemissen> <level> <parent> <goal-stack> )
<Praemissen>     = ( <p1> <p2> ... )

<choice-stack>   = ( <choice-point>+ )
<choice-point>   = ( <trail>
                     <redo-fail-list>
                     <database-left>
                     <level-of-db>
                     <goal>
                     <level-of-goal>
                     <goal-stack>
                     <clause> )
```

Wird ein Ziel mit einer Klausel resolviert, so werden auf dem "goal-stack" die Praemissen der Klausel, der "level-of-db" dieser Klausel und das Parent-Goal abgelegt. Das Parent-Goal ist instanziert und umbenannt und wird nur fuer den Tracer benoetigt. Die Praemissen sind unverandert aus der Datenbasis uebernommen.

Beim Resolutionsschritt wird ausserdem ein Element auf dem "choice-stack" abgelegt. Ein solches Element entspricht einer noch offenen Backtrackmoeglichkeit auf dem "goal" mit "database-left" als alternativer Klauselmenge. Werden Resolutionen durchgefuehrt, fuer die es keine weiteren Alternativen mehr gibt, so werden die dabei erzeugten Bindungen in das oberste Stackelement hinzugefuegt. Damit das immer moeglich ist, existiert immer mindestens ein Element auf dem Stack.

In der <redo-fail-list> werden Informationen fuer den Tracer gesammelt, die beim Backtracking die Ausgabe von "redo" und "fail" Signalen veranlassen.

2.1.4. Kontrollfluss in der Funktion "prove"

Die Funktion "prove" besteht aus einem einzigen "prog" Ausdruck, in dem der Kontrollfluss durch "go" Anweisungen gesteuert wird. Zum besseren Verstaendnis wird im folgenden der Kontrollfluss innerhalb des "prog" Ausdrucks mit dem Kontrollfluss in den Funktionen "and-process" und "or-process" des LISPLOG.1-Interpreters verglichen.

Bei dem Entwurf von strukturierten Programmen ist eine der Grundannahmen, dass sich ein Programm in kleinere Bloecke zerlegen laesst, die alle genau einen Eingang und einen Ausgang haben. Das vielleicht auffaelligste Merkmal eines Backtracking-Programmes (wie es der LISPLOG-Interpreter ist), ist die Tatsache, dass alle Bloecke einen Eingang und z w e i Ausgaenge haben. Dabei entspricht ein Ausgang dem gelungenen Resolutionsschritt ("Exit"), waehrend der andere Ausgang dem misslungenen Resolutionsschritt mit nachfolgendem Backtracking ("Fail") entspricht.

Im LISPLOG.1-Interpreter wird der "Exit" Ausgang durch einen (ko-)rekursiven Aufruf, der "Fail" Ausgang durch ein Funktionsende ("return") realisiert. Dies hat den merkwuerdigen Effekt, dass das erfolgreiche Ende eines LISPLOG-Beweises auf der tiefsten Aufrufstufe erfolgt und nicht wie in anderen LISP-Funktionen nach Beendigung aller Aufrufe.

Der LISPLOG.2-Interpreter realisiert beide Ausgaenge eines Blocks durch Spruenge ("go"). Dabei entspricht ein Sprung zur Marke "and-label" einem rekursiven Aufruf in LISPLOG.1, ein Sprung zu "redo-on-choice-stack" entspricht einer "return" Operation in LISPLOG.1.

Die Bloecke, die jeweils diese zwei Ausgaenge haben, sind in beiden Versionen (LISPLOG 1 und LISPLOG.2) praktisch identisch. In LISPLOG.2 sind sie lediglich aus Gruenden der Uebersichtlichkeit mit eigenen Marken versehen ("atomic-goal", "primitive", "lisp-defined" und "user-defined").

Die Marke "or-label" in LISPLOG.2 wird fuer die Schleife benoetigt, die in LISPLOG.1 durch rekursive Aufrufe des "or-process" entsteht. Diese Marke dient gleichzeitig als Einstiegspunkt beim Backtracking.

2.2. Speicherung des LISPLOG.2-Environment

Der LISPLOG.2-Interpreter speichert das Environment in einer globalen Variable. Diese Variable ist ein Array von Assoziationslisten, auf das sehr effizient zugegriffen werden kann. Fuer die Operationen Einfuegen, Loeschen und Suchen wird hier eine annaehernd konstante Zeitkomplexitaet erreicht. Nachfolgend werden die benutzten Datenstrukturen beschrieben, und der prinzipielle Ablauf der Operationen dargestellt.

2.2.1. Datenstrukturen

Benutzt werden Variablen, die von den folgenden Typen sind :

```
<INDEX-1>, <INDEX-2>   = 0..127

<LEVEL>                = ( <INDEX-2> <INDEX-1> )

<VARIABLE-OHNE-INDEX>  = ( "?" <NAME> )

<VARIABLE-MIT-INDEX>   = ( "?" <NAME> <INDEX-2> <INDEX-1> )

<ENVIRONMENT>          = HUNK [ <INDEX-1> ] OF
                            HUNK [ <INDEX-2> ] OF
                                 LIST OF
                                      ( <NAME> . <VALUE> )

<TRAIL>                = DOTTED LIST OF ( <NAME> . <VALUE> )
```

Bemerkungen zur Notation :

<x>	bezeichnet den Datentyp x
<x> = y	y ist die Definiton des Datentyps x
0..127	Integerzahlen von 0 bis 127
( ) (.)	Listen bzw. Dotted Pairs
"?"	Die Konstante ?
HUNK [<i>] OF <x>	ein HUNK mit Indexbereich i und Werten x
LIST OF <x>	eine LISP-Liste mit Elementen x
DOTTED LIST OF <x>	eine Dotted-List mit Elementen x

Im ENVIRONMENT ist eine Menge von Bindungen gespeichert. Jede Bindung besteht jeweils zwischen einer VARIABLE-MIT-INDEX und einem VALUE (beliebiger LISP-Ausdruck). HUNK ist eine FRANZ-LISP Datenstruktur, die einem Array in PASCAL entspricht (s. FRANZ-LISP Manual). Eine Variable vom Typ ENVIRONMENT wird bei jedem Aufruf der Funktion "prove" angelegt.

Ein TRAIL endet mit einem Atom (i.a. "t"), damit ein leeres TRAIL nicht "nil" ist. Dies entspricht der Verwendung von "bottom-of-environment" im LISPLOG.1-Interpreter.

2.2.2. Operationen

Ein Zugriff auf eine Bindung einer VARIABLE-MIT-INDEX erfolgt dadurch, dass die Komponenten der Variable das ENVIRONMENT indexieren. Dabei wird fuer die Komponente NAME eine lineare Suche in einer Assoziationsliste durchgefuehrt (Funktion "env-get").

Wird eine neue Bindung aufgenommen, so wird die entsprechende Assoziationsliste erweitert. Dabei wird das neue NAME-VALUE Paar auch in ein TRAIL aufgenommen. Das Dotted Pair NAME-VALUE wird dabei nur einmal angelegt (shared structure). Ueber das TRAIL ist daher direkt die Bindung im ENVIRONMENT erreichbar, was beim Loeschen ausgenutzt wird (Funktion "env-add").

Das Loeschen von Bindungen erfolgt unter Benutzung des beim Einfuegen erzeugten TRAIL. Dabei wird fuer jedes Element des TRAIL das Feld NAME durch einen Zeiger auf sich selbst ersetzt Dadurch wird auch das ensprechende Feld im ENVIRONMENT modifiziert (durch shared structure). Die Implementierung des Zugriffs auf eine Bindung bewirkt, dass die Bindung durch diese Modifikation geloescht wird (Funktionen "env-rem" und "unused-env-cell").

Beim Loeschen entstehen somit in den Assoziationslisten unbenutzte NAME-VALUE Paare, die beim Einfuegen neuer Bindungen wieder belegt werden um die Laenge der Assoziationslisten klein zu halten (Funktion "env-add-1").

Ein ENVIRONMENT besteht aus maximal 128 + 1 HUNK Elementen. Diese werden dynamisch erzeugt um Speicherplatz zu sparen. Am Anfang existiert nur der HUNK auf der ersten Ebene sowie der HUNK mit INDEX-1 = 0 auf der zweiten Ebene (Funktionen "env-init" und "level-next").

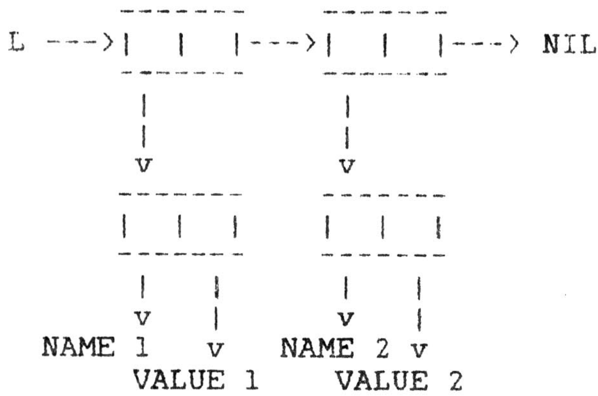
Die Operationen auf dem ENVIRONMENT haben eine annaehernd konstante Zeitkomplexitaet, wenn man annimmt, dass nur wenige Variablen auf einem Level existieren. Diese Annahme ist gerechtfertigt, weil die Anzahl der Variablen auf einem Level der Anzahl der Variablen einer auf diesem Level benuzten Klausel entspricht. In LISPLOG-Klauseln kommen aber selten mehr als etwa 5 Variablen vor.

Der INDEX Bereich 0..127 ist durch die FRANZ-LISP Datenstruktur HUNK bedingt. Der Wert 127 ist dabei aber nicht signifikant.

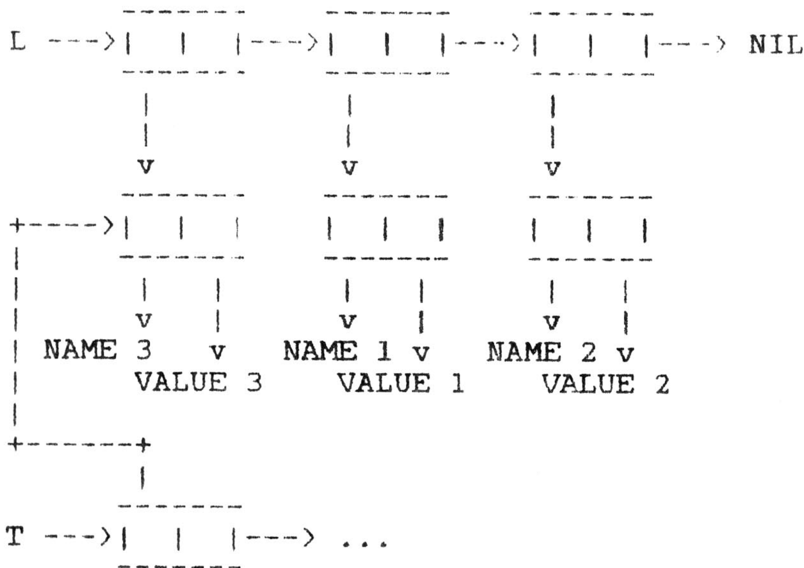
2.2.3. Schematische Darstellung von Einfuegen und Loeschen

Dargestellt werden eine Assoziationsliste aus einem ENVIRONMENT (L) und ein TRAIL (T).

mit zwei Bindungen in der Liste L



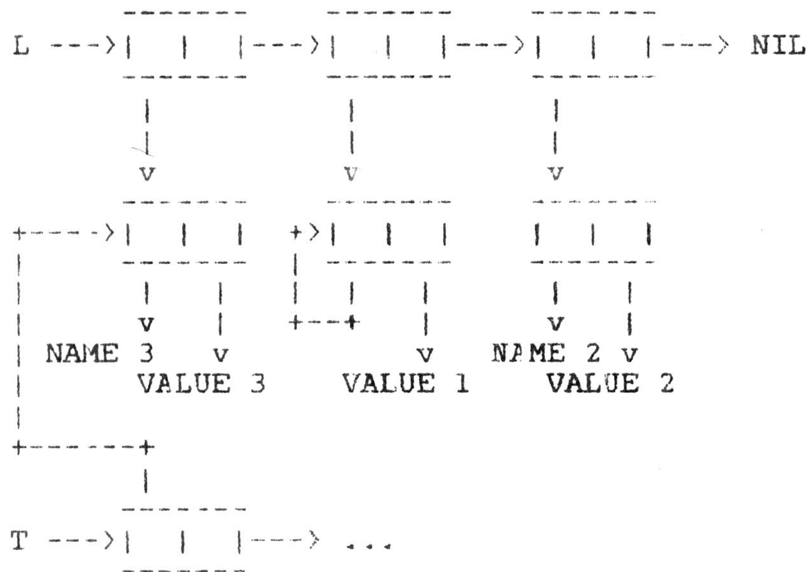
nach Einfuegen einer weiteren Bindung



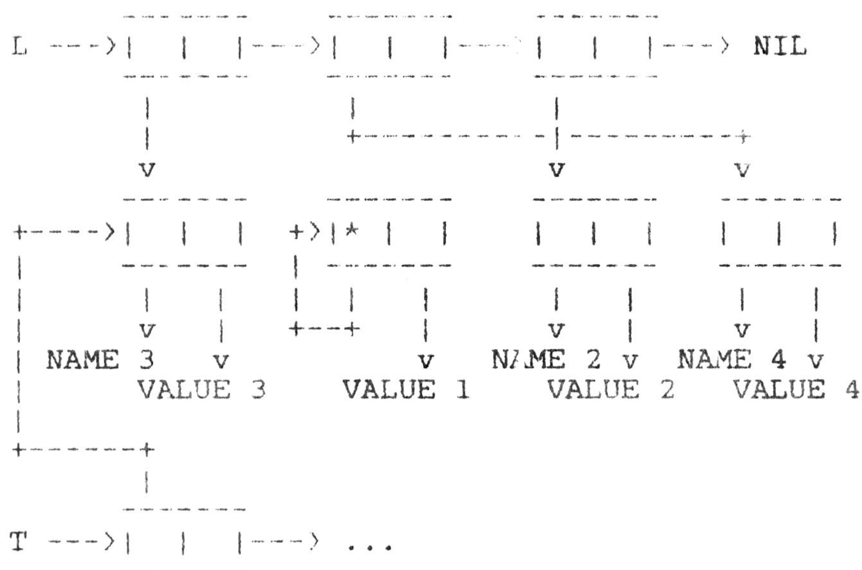
Der Ausschnitt aus einem TRAIL ist nur fuer die neue Bindung gezeigt, ein solcher Zeiger existiert aber fuer jede Bindung. Im naechsten Bild ist jetzt die Bindung NAME 1 geloescht.

Dargestellt werden eine Assoziationsliste aus einem ENVIRONMENT (L) und ein TRAIL (T).

Die Assoziationslisten enthaelt zwei benutzte und eine unbenutze Listenzelle.



Wird jetzt eine neue Bindung aufgenommen, so wird die freie Stelle in der Liste benutzt.



Die mit "*" markierte Zelle ist jetzt Garbage.

2.2.4. Variablen in LISP-Aufrufen von LISPLOG

Wird von LISPLOG aus eine LISP-Funktion aufgerufen, so koennen die Argumente der LISP-Funktion auch ungebundene Variablen enthalten (z.B. assertieren von Regeln, Aufruf von n-solutions u.s.w.). Das Problem dabei ist, dass im LISPLOG.2-Interpreter alle Variablen durch Name und Level gekennzeichnet sind, waehrend Daten, die der Interpreter aus dem LISP-Environment uebernimmt (z.B. aus der Klauselbasis oder als Top-Level-Ziele) nur durch einen Name gekennzeichnet sein koennen (der Interpreter fuehrt darauf ja ein "rename" durch).

Dieses Problem trat im LISPLOG.1 nicht direkt zutage, da dort einfach die Anzahl der Levelindizes bei jedem solchen Schritt erhoeht wurde. Im LISPLOG.2-Interpreter ist das nicht mehr so moeglich, stattdessen wird der Level beim Aufruf von LISP-Funktionen aus den Variablen eliminiert. Da dabei Variablen wie "(? x 2)" und "(? x 3)" zusammenfallen wuerden, gibt es eine Funktion "unique-name" die dann neue Variablennamen generiert.

3. Laufzeitvergleiche

Im folgenden werden Ergebnisse von Laufzeitvergleichen zwischen den LISPLOG Versionen LISPLOG.1 und LISPLOG.2 dargestellt. Beide Versionen sind sowohl in FRANZ-LISP unter UNIX BSD 4.2 als auch in COMMON-LISP auf einer SYMBOLICS 3640 implementiert. In den Vergleich einbezogen wird ausserdem die CPROLOG Implementierung unter UNIX BSD 4.2. CPROLOG ist sicher keine besonders schnelle Implementierung (kein Compiler), es wurde in den Vergleich aber aufgenommen, weil bislang keine bessere PROLOG Implementierung verfuegbar war.

3.1. Vorbemerkungen

Die Messung der Laufzeit von Programmen ist nicht unproblematisch. Insbesondere beim Vergleich verschiedener Interpreter hat die Auswahl der Programme, mit denen die Interpreter getestet werden einen erheblichen Einfluss auf das Ergebnis des Vergleichs. Dieser Effekt ist ebenso von Vergleichen verschiedener Prozessoren oder Betriebssyteme bekannt und Spoetter behaupten, man koenne fuer jedes gewuenschte Ergebnis eine geeignete Auswahl von Testprogrammen finden. Die weiter unten angegebenen Ergebnisse sollten daher immer mit Blick auf die verwendeten, zwar breit gestreuten, aber trotzdem nicht fuer jede Anwendung repraesentativen Testprogramme bewertet werden.

3.2 Verwendete Testprogramme

Fuer die Vergleiche wurden sowohl spezielle Benchmark-Programme als auch Anwendungen, die nicht zum Zwecke des Benchmarks erstellt wurden, benutzt. Die Benchmark-Programme wurden vom ECRC Muenchen [ECRC 1986] uebernommen. Sie testen gezielt einige Aspekte von PROLOG Interpretern bzw. Compilern, und geben somit einen genauen Einblick in die Interpreter. Diese Testprogramme zeigen daher sehr deutlich, welche Operationen besonders gut und welche besonders schlecht von dem jeweiligen Interpreter ausgefuehrt werden koennen. Die Anwendungsprogramme stammen von verschiedenen Mitarbeitern der LISPLOG Arbeitsgruppe. Das Programm zum symbolischen Differenzieren stammt von J. Herr, das Schachendspiel wurde aus [van Emden 1982] entnommen und von M. Dahmen in LISPLOG realisiert. Diese Programme testen keinen speziellen Aspekt sondern den PROLOG Interpreter als ganzes.

3.3 Masseinheit und Messverfahren

Die Ergebnisse in der Tabelle sind in LIPS (Logical Inferences per second) angegeben. Die Ergebnisse fuer den CPROLOG Interpreter unter UNIX BSD 4.2 wurden ungeprueft aus [ECRC 1986] uebernommen. Die Messungen fuer die LISPLOG-Interpeter unter UNIX wurden nachts nach 23 Uhr vorgenommen, um moeglichst keine Verfaelschungen der Werte durch den Mehrbenutzerbetrieb zu erhalten. Alle Messungen wurden mehrfach durchgefuehrt, die Abweichungen vom angegebenen Mittelwert lagen unter 5 %.

3.4. Messergebnisse

(1) CPROLOG unter UNIX auf VAX 11/785 laut [ECRC 1986]
(1a) CPROLOG unter UNIX auf VAX 11/750 (eigene Messungen)
(2) LISPLOG.2 in FRANZ-LISP unter UNIX auf VAX 11/750
(3) LISPLOG.2 in COMMON-LISP auf Symbolics
(4) LISPLOG.1 in FRANZ-LISP unter UNIX auf VAX 11/750
(5) LISPLOG.1 in COMMON-LISP auf Symbolics

Werte in LIPS

Aspekt des Testprogramms	(1)	(2)	(3)	(4)	(5)
Benchmarks :					
simple calls	7700	320	2730	SF	1060
non-deterministic					
creation of choice points	4020	30	419	7	11
deep backtracking	2320	67	174	10	28
shallow backtracking	3530	110	338	15	45
handling of environments	4230	38	249	11	5
indexing	1240	92	344	32	149
unification					
list construction via unification	6000	262	1058	30	173
list matching via unification	4200	95	315	28	149
structure construction	6000	252	894	30	171
structure matching	4300	28	112	17	80
match a nested structure	170	3	30	9	160
general unification	170	2	20	26	88
Mittelwert ueber alle Bechmarks	3560	108	557	20	177
Relation	183	5	27	1	9
Anwendungen :					
Symbolisches Differenzieren	(1a)				
bei 231 Calls		64	316	11	31
bei 398 Calls		77	316	SF	SF
Schachendspiel					
bei 810 Calls	659	77	404	15	41
bei 1204 Calls	627	70	249	15	40
bei 4910 Calls	692	61	322	13	38
bei 11215 Calls	628	63	308	15	40
bei 12567 Calls	685	66	304	18	47
bei 16425 Calls	608	65	301	17	43
Mittelwert ueber alle Anwendungen	633	68	315	15	40
Relation	42	5	21	1	2.7

SF = Programmabruch durch Stack Overflow
leere Felder = Messung nicht durchgefuehrt

3.5. Anmerkungen

Im LISPLOG.2-Interpreter wird, besonders bei laengeren Beweisen, ein Anteil von etwa 45% der Gesamtlaufzeit fuer Speicherbereinigung (Garbage Collection) benoetigt. Beim LISPLOG.1-Interpreter ist dieser Wert numerisch etwa gleich gross, macht aber nur etwa 10% der Gesamtlaufzeit aus. Beruecksichtigt man dieser, durch das LISP-System festgelegten Anteil nicht, so ist der LISPLOG.2-Interpreter etwa 8-mal so schnell wie der LISPLOG.1-Interpreter. Fuer den Anwender von LISPLOG ist natuerlich unerheblich wofuer der Interpreter die Laufzeit benoetigt, daher wurden die Zeiten fuer Speicherbereinigung in den obigen Tabellen mit beruecksichtigt.

Durch den Zeitbedarf fuer Speicherbereinigung ist auch zu erklaeren, wieso die Unterschiede zwischen LISPLOG.1 und LISPLOG.2 auf der Symbolics anders sind, als auf der VAX. Auf der VAX ist LISPLOG.2 etwa 5-mal schneller als LISPLOG.1, auf der Symbolics dagegen ist LISPLOG.2 etwa 8-mal schneller als LISPLOG.1. Die Ursache ist darin zu sehen, dass die Symbolics die Speicherbereinigung konkurrent zur eigentlichen Berechnung durchfuehrt (Garbage Collection on the fly), waehrend bei der VAX die Programmausfuehrung unterbrochen werden muss, um die Speicherbereinigung durchzufuehren.

Dieser Effekt ist am besten bei den Anwendungen zu erkennen, da diese bei allen Interpretern mit exakt gleichen Anfragen getestet wurden. Bei den Benchmarkprogrammen dagegen muesste die Laenge der Beweise fuer LISPLOG 1 deutlich gekuerzt werden, um einen Ueberlauf des LISP-Stacks zu vermeiden. Aus diesem Grund sind auch die Werte fuer LISPLOG.1 noch guenstiger, als sie bei laengeren Beweisen waeren. Bei LISPLOG.2 wurden meist die Beweislaengen gewaehlt, die auch fuer den CPROLOG-Interpreter gewaehlt wurden. Daher sind die Werte CPROLOG zu LISPLOG.2 besser vergleichbar.

Fuer die letzten beiden Benchmarks faellt der Wert fuer den LISPLOG.1-Interpreter extrem guenstig aus. Dies ist dadurch zu erklaeren, dass bei diesem Benchmark sehr grosse Terme unifiziert werden, die aber keine Variablen enthalten. Diese Terme sind somit "equal" im Sinne der gleichnamigen LISP-Funktion. Da der LISPLOG.1-Interpreter in der Unifikation zunaechst diesen Test auf Gleichheit vornimmt, erreicht er in diesem Spezialfall eine untypisch hohe Geschwindigkeit. Im LISPLOG.2-Interpreter ist dieser Test nicht eingebaut, da er im allgemeinen keinen positiven Effekt bringt (vgl. dazu 1.2.5).

4. Literatur

[Boley & Kammermeier et al. 1985]
H. Boley, F. Kammermeier u. die LISPLOG-Gruppe:
LISPLOG: Momentaufnahmen einer
LISP/PROLOG-Vereinheitlichung.
Universitaet Kaiserslautern, FB Informatik,
MEMO SEKI-85-03, August 1985.
Short version in: B. Nebel (Ed.): Papiere zum
Workshop Logisches Programmieren und Lisp.
TU Berlin, FB Informatik, KIT-REPORT 31, Dec. 1985.

[ECRC 1986]
J.C. SYRE et al.:
Benchmark Programs for PROLOG Systems.
ECRC (European Computer-industry Research Center),
Muenchen, April 1986
Notes net.lang.prolo 29.4.1986
Kommunikation ueber UUCP
Adresse : ecrcvax!jclaude

[Herr 1986]
J. Herr:
LISPLOG Compiler und Interpeter.
Universitaet Kaiserslautern, FB Informatik,
SEKI WORKING PAPER, Juni 1986

[van Emden 1982]
M.H. van Emden:
Chess Endgame Advice
in:
D. Michie (Ed.) :
Introductory Readings in Expert Systems.
Gordow and Breach Science Publishers,
New York 1982

Anhang 1 : Programmlistings des LISPLOG.2-Interpreters

Datei : lisplog.2.1

```
; utilities.2.1
; allgemein benutzte Funktionen nicht lokal
; localf some mapcardot setcons remlist union without n-spaces

; environ.2.1
(declare (localf level-null level-next
                 env-init env-add env-add-1 env-rem env-get
                 ultimate-assoc ultimate-inst unify unify-rename
                 rename-variables rename-inst
                 top-level-bindings-p
                 get-top-level-bindings print-top-level-bindings
                 unique-name unique-name-1 unique-name-2
                 concat-trail ))
; trace-inst trace-unify
; nicht local, Schnittstelle zum Tracer

; lispeval.2.1
(declare (localf lisp-predicates application-p
                 quote-application expand-macro))
; contain-freevars quote-lambda-arg quote-nlambda-arg discipline
; nicht local, da Mapping mit diesen Funktionen

; primitives.2.1
(declare (localf prolog-primitive execute-arith
                 execute-not execute-is))

; prove.2.1
(declare (localf pop-redo-fail))

; assert.1.1
(declare (localf ass-1 assertlst pp-clauses pp-predlst
                 retr-1 retractlst retract-plst save-dblst
                 get-predlst get-clauses)
          (nlambda ass assertimp listing rex retractimp
                 abolish consult tell edit ))

; interface.2.1
(declare (localf commands y-or-n-p do-external-help print-help))
; do-exit-lisplog do-help
; nicht local, da eval mit diesen Funktionen
```



```
(declare (special arith-predicates primitives
                  lisp-coms lisplog-top-level-prompt leave-prolog
                  prinlength prinlevel
                  toplevel-flag quote-lisp-calls
                  unique-name-list environment
                  predicates tracemode breakmode indent)
         (macros t))

(include absynt.2.1)
(include utilities.2.1)
(include environ.2.1)
(include primitives.2.1)
(include lispeval.2.1)
(include prove.2.1)
(include interface.2.1)
(include extern.2.1)
(include assert.1.1)

(setq predicates nil)
(setq toplevel-flag nil)
(setq quote-lisp-calls t)
(lisplog-start-message)
```

Datei : prove.2.1

```
;-----------------------------------------------------------------
; Zugriff auf LISPLOG Datenbasis
;-----------------------------------------------------------------

(defmacro is-user-defined (goal)
  `(get (first ,goal) 'clauses))

(defmacro get-database (goal)
  `(get (first ,goal) 'clauses))

;-----------------------------------------------------------------
; Kommunikation mit Tracer
;-----------------------------------------------------------------

(defmacro aktiv-tracer (goal)
   'tracemode)

;-----------------------------------------------------------------
; Operationen auf Stacks
;-----------------------------------------------------------------

(defmacro init-stack (stack)
  `(setq ,stack nil))

(defmacro pop-stack (stack)
  `(setq ,stack (rest ,stack)))

(defmacro top (stack)
  `(first ,stack))

(defmacro empty (stack)
  `(null ,stack))

;-----------------------------------------------------------------
; Spezielle Stack Operationen
;-----------------------------------------------------------------

(defmacro push-choice-point (elem choice-stack)
  `(setq ,choice-stack (cons ,elem ,choice-stack)))

(defmacro push-trail (trail choice-stack)
  `(rplaca (top ,choice-stack)
           (concat-trail ,trail (first (top ,choice-stack)))))
```



```
(defmacro push-goals (new-goals level parent goal-stack)
  `(if (aktiv-tracer ,parent)
       (setq ,goal-stack (list ,new-goals ,level ,parent ,goal-stack))
       (setq ,goal-stack (list ,new-goals ,level nil ,goal-stack))))

(defmacro pop-goal (goal-stack)
  `(setq ,goal-stack (cons (rest (first ,goal-stack))
                           (rest ,goal-stack))))

(defmacro pop-parent (goal-stack)
  `(setq ,goal-stack (fourth ,goal-stack)))

;-----------------------------------------------------------------
; Stack Operationen fuer Trace Erzeugung
;-----------------------------------------------------------------

(defmacro push-redo (goal choice-stack)
  `(if (aktiv-tracer ,goal)
       (rplaca (rest (top ,choice-stack))
               (cons 'redo
                     (cons ,goal (second (top ,choice-stack)))))))

(defmacro push-fail (goal choice-stack)
  `(if (aktiv-tracer ,goal)
       (rplaca (rest (top ,choice-stack))
               (cons 'fail
                     (cons ,goal (second (top ,choice-stack)))))))

(defun pop-redo-fail (redo-fail-list)
   (cond ((null redo-fail-list))
         ((equal 'redo (first redo-fail-list))
          (box-redo (second redo-fail-list))
          (pop-redo-fail (rest (rest redo-fail-list))))
         ((equal 'fail (first redo-fail-list))
          (box-fail (second redo-fail-list))
          (pop-redo-fail (rest (rest redo-fail-list))))))

;-----------------------------------------------------------------
; Operationen nach Ende des Beweises
;-----------------------------------------------------------------

(defmacro save-solution (solutions result-term)
  `(setq ,solutions
         (cons (cond (,result-term (rename-inst ,result-term
                                                 (level-null)))
                     (t            (get-top-level-bindings)))
               ,solutions)))
```

```
(defmacro end-of-prove (anzahl solutions result-term)
  `(cond ((equal ,anzahl 1)
          (save-solution ,solutions ,result-term)
          (return ,solutions))
         ((and (numberp ,anzahl) (greaterp ,anzahl 1))
          (save-solution ,solutions ,result-term)
          (setq ,anzahl (sub1 ,anzahl))
          (go redo-on-choice-stack))
         (t (patom "success")
            (terpr)
            (print-top-level-bindings)
            (if (and (top-level-bindings-p)
                     (y-or-n-p "More? (y or n) "))
                (go redo-on-choice-stack)
                (return t)))))

;-----------------------------------------------------------------
; Hauptfunktion des LISPLOG Beweisers
; Argumente der Funktion :
; 1) Liste der zu beweisenden Ziele
;    Variablen ohne Levelindex
; 2) nil  => interaktiv d.h. Benutzer fordert weitere
;            Loesungen an
;    Zahl => es werden hochestens soviel Loesungen
;            bestimmt (keine Interaktion)
; 3) nil  => Loesung wird repraesentiert durch das
;            Environment
;   sonst => Loesung wird repraesentiert durch
;            instanziierung dieses Arguments
;-----------------------------------------------------------------

(defun prove (top-level-goal-list anzahl result-term)
  (prog (goal-stack
         choice-stack
         environment
         solutions
         trail
         database-left
         level-of-db
         goal
         level-of-goal
         clause
         )
```



```
        ;-------------------------------------------------------
        ; Initialisieren der lokalen Variablen
        ;-------------------------------------------------------

        (init-stack goal-stack)
        (init-stack choice-stack)
        (setq solutions nil)
        (env-init)
        (setq level-of-db (level-null))

        (push-goals top-level-goal-list level-of-db nil goal-stack)
        (push-choice-point (list nil
                                 nil
                                 nil
                                 (level-null)
                                 nil
                                 (level-null)
                                 goal-stack
                                 nil)
                           choice-stack)

;-----------------------------------------------------------------
; Nimm naechstes Ziel vom Stack und beweise
;-----------------------------------------------------------------
and-label

        (cond ((empty goal-stack)
               (end-of-prove anzahl solutions result-term)))

        (cond ((null (top goal-stack))
               (cond ((third goal-stack)
                      (box-exit (third goal-stack))
                      (push-redo (third goal-stack)
                                 choice-stack)))
               (pop-parent goal-stack)
               (go and-label)))

        (setq level-of-goal (second goal-stack))
        (setq goal (rename-inst (first (top goal-stack))
                                level-of-goal))
        (pop-goal goal-stack)

        (box-call goal)

        ;-------------------------------------------------------
        ; Analysiere Ziel  (Befehlsverteiler)
        ;-------------------------------------------------------
```

```
        (cond ((variable-p goal))
              ((atom goal) (go atomic-goal))
              ((not (consp goal)))
              ((variable-p (first goal)))
              ((is-user-defined goal) (go user-defined))
              ((is-primitive goal) (go primitive))
              ((is-lisp-defined goal) (go lisp-defined)))

        (box-undef goal)
        (box-fail goal)
        (go redo-on-choice-stack)

;-------------------------------------------------------------
atomic-goal

        (cond ((not goal)
               (box-fail goal)
               (go redo-on-choice-stack)))

        (box-exit goal)
        (go and-label)

;-------------------------------------------------------------
primitive

        (setq trail (prolog-primitive goal))

        (cond ((null trail)
               (box-fail goal)
               (go redo-on-choice-stack)))

        (push-trail trail choice-stack)
        (box-exit goal)
        (go and-label)

;-------------------------------------------------------------
lisp-defined

        (cond ((not (lisp-predicates goal))
               (box-fail goal)
               (go redo-on-choice-stack)))

        (box-exit goal)
        (go and-label)

;-------------------------------------------------------------
user-defined

        (setq level-of-db (level-next level-of-db))
        (setq database-left (get-database goal))
```



```
;-------------------------------------------------------------
; Entry Punkt fuer Redo Operation
;-------------------------------------------------------------
or-label

        (cond ((null database-left)
               (box-fail goal)
               (go redo-on-choice-stack)))

        (setq clause (first database-left))
        (setq database-left (rest database-left))
        (setq trail
              (unify-rename (s-conclusion clause)
                            level-of-db
                            goal
                            t))

        (cond ((null trail) (go or-label)))

        ;-----------------------------------------------------
        ; Anwendbare Klausel gefunden
        ;-----------------------------------------------------

        ; initialer Cut ?

        (cond ((cut-p clause)
               (box-cut goal clause database-left)
               (setq database-left nil)))

        ;-----------------------------------------------------
        ; sichere aktuellen Zustand des Beweises
        ;-----------------------------------------------------

        (cond (database-left
               (push-choice-point (list trail
                                        nil
                                        database-left
                                        level-of-db
                                        goal
                                        level-of-goal
                                        goal-stack
                                        clause)
                                  choice-stack))
              (t
               (push-trail trail choice-stack)
               (push-fail goal choice-stack)))
        (push-goals (s-premises clause) level-of-db goal goal-stack)
        (go and-label)
```

```
;---------------------------------------------------------------
; Backtracking : aktiviere letzten choice-point
;---------------------------------------------------------------
redo-on-choice-stack

      (env-rem              (first  (top choice-stack)))
      (pop-redo-fail        (second (top choice-stack)))
      (setq database-left   (third  (top choice-stack)))
      (setq level-of-db     (fourth (top choice-stack)))
      (setq goal            (fifth  (top choice-stack)))
      (setq level-of-goal   (sixth  (top choice-stack)))
      (setq goal-stack      (seventh (top choice-stack)))
      (setq clause          (nth 7 (top choice-stack)))
      (pop-stack choice-stack)

      (cond ((empty choice-stack)
             (return solutions)))

      (cond ((box-cut-p goal clause database-left)
             (go redo-on-choice-stack)))

      (go or-label)

      ))
```

Datei : absynt.2.1

```
(defmacro first (x)
  `(car ,x))

(defmacro second (x)
  `(cadr ,x))

(defmacro third (x)
  `(caddr ,x))

(defmacro fourth (x)
  `(cadddr ,x))

(defmacro fifth (x)
  `(car (cddddr ,x)))

(defmacro sixth (x)
  `(cadr (cddddr ,x)))

(defmacro seventh (x)
  `(caddr (cddddr ,x)))

(defmacro rest (x)
  `(cdr ,x))

(defmacro consp (x)
  `(dtpr ,x))

(defmacro if (condition then-do else-do)
  `(cond (,condition ,then-do) (t ,else-do)))
```


Datei : utilities.2.1

```
(defun mapcardot (funct dotlst)
   (cond ((atom dotlst) dotlst)
         (t
          (cons (funcall funct (first dotlst))
                (mapcardot funct (rest dotlst))))))

(defun some (funct lst)
   (cond ((null lst) nil)
         ((funcall funct (first lst)) lst)
         (t (some funct (rest lst)))))

(defun remlist (l1 l2)
   (cond ((null l1) l2)
         (t (remlist (rest l1) (remove (first l1) l2)))))

(defun setcons (elem set)
   (cond ((member elem set) set) (t (cons elem set))))

(defun union (s1 s2)
   (cond ((null s1) s2)
         ((member (first s1) s2) (union (rest s1) s2))
         (t (cons (first s1) (union (rest s1) s2)))))

(defun without (x y)
   (cond ((null x) nil)
         ((equal (first x) y) (rest x))
         (t (cons (first x) (without (rest x) y)))))

(defmacro catch-all-errors (expr if-error)
  `(let ((result (errset ,expr)))
        (if result (first result) ,if-error)))

(defun n-spaces (n)
   (cond ((lessp 0 n) (patom " ") (n-spaces (sub1 n)))))
```

Datei : environ.2.1

```
(defmacro cut-p (clause)
  `(eq (first (first ,clause)) 'cut))

(defmacro s-conclusion (clause)
  `(cond ((cut-p ,clause) (second (first ,clause)))
         (t (first ,clause))))

(defmacro s-premises (clause)
  `(rest ,clause))

(defmacro variable-p (expr)
  `(and (consp ,expr) (eq (first ,expr) '?)))

(defmacro anonym-var-p (expr)
  `(eq ,expr 'ID))

(defmacro level-of (var)
  `(and (equal (length ,var) 4)
        (plus (third ,var) (times (fourth ,var) 128))))

(defmacro name-of (var)
  `(second ,var))

(defmacro unused-env-cell (cell)
  `(eq (first ,cell) ,cell))

(defun level-null nil
   (list 0 0))

(defun level-next (level)
   (cond ((equal 127 (first level))
          (rplacx (addl (second level))
                  environment
                  (makhunk 128))
          (list 0 (addl (second level))))
         (t (cons (addl (first level)) (rest level)))))

(defun env-init nil
   (setq environment (makhunk 128))
   (rplacx 0 environment (makhunk 128)))
```



```
(defun env-add (var value trail)
   (let ((curr-hunk (cxr (fourth var) environment))
         (curr-index (third var))
         (new-var-value (cons (name-of var) value)))
        (cond
         ((env-add-1 new-var-value
                     (cxr curr-index curr-hunk))
          (rplacx curr-index
                  curr-hunk
                  (cons new-var-value
                        (cxr curr-index curr-hunk)))))
        (cons new-var-value trail)))

(defun env-add-1 (new-var-value env-list)
   (cond ((null env-list))
         ((unused-env-cell (first env-list))
          (rplaca env-list new-var-value)
          nil)
         (t (env-add-1 new-var-value (rest env-list)))))

(defun env-rem (trail)
   (cond
    ((consp trail)
     (rplaca (first trail) (first trail))
     (env-rem (rest trail)))))

(defun env-get (var)
   (let ((curr-hunk (cxr (fourth var) environment)))
        (cond
         (curr-hunk
          (assq (name-of var) (cxr (third var) curr-hunk))))))

(defun ultimate-assoc (x)
   (cond ((variable-p x)
          (let ((binding (env-get x)))
               (cond ((null binding) x)
                     (t (ultimate-assoc (rest binding))))))
         (t x)))

(defun ultimate-inst (x)
   (cond ((atom x) x)
         ((variable-p x)
          (let ((binding (env-get x)))
               (cond ((null binding) x)
                     (t (ultimate-inst (rest binding))))))
         (t
          (cons (ultimate-inst (first x))
                (ultimate-inst (rest x))))))
```

```
(defun unify (x y trail)
   (let ((x (ultimate-assoc x)) (y (ultimate-assoc y)))
        (cond ((eq x y) trail)
              ((or (anonym-var-p x) (anonym-var-p y)) trail)
              ((variable-p x)
               (cond ((equal x y) trail)
                     (t (env-add x y trail))))
              ((variable-p y) (env-add y x trail))
              ((and (numberp x) (equal x y)) trail)
              ((and (atom x) (equal x y)) trail)
              ((or (atom x) (atom y)) (env-rem trail))
              (t
               (let ((new-trail
                      (unify (first x) (first y) trail)))
                    (and new-trail
                         (unify (rest x)
                                (rest y)
                                new-trail))))))))

(defun unify-rename (x level-of-x y trail)
   (let ((y (ultimate-assoc y)))
        (cond ((eq x y) trail)
              ((or (anonym-var-p x) (anonym-var-p y)) trail)
              ((variable-p x)
               (unify (cons '?
                            (cons (second x) level-of-x))
                      y
                      trail))
              ((variable-p y)
               (env-add y
                        (rename-variables x level-of-x)
                        trail))
              ((and (numberp x) (equal x y)) trail)
              ((and (atom x) (equal x y)) trail)
              ((or (atom x) (atom y)) (env-rem trail))
              (t
               (let ((new-trail
                      (unify-rename (first x)
                                    level-of-x
                                    (first y)
                                    trail)))
                    (and new-trail
                         (unify-rename (rest x)
                                       level-of-x
                                       (rest y)
                                       new-trail))))))))
```



```
(defun rename-variables (term level-of-term)
   (cond ((atom term) term)
         ((variable-p term)
          (cons '?
                (cons (name-of term) level-of-term)))
         (t
          (cons (rename-variables (first term)
                                  level-of-term)
                (rename-variables (rest term)
                                  level-of-term)))))

(defun rename-inst (term level-of-term)
   (cond ((atom term) term)
         ((variable-p term)
          (ultimate-inst
           (cons '?
                 (cons (name-of term) level-of-term))))
         (t
          (cons (rename-inst (first term) level-of-term)
                (rename-inst (rest term) level-of-term)))))

(defun top-level-bindings-p nil
   (cxr 0 (cxr 0 environment)))

(defun get-top-level-bindings nil
   (remove nil
           (mapcar '(lambda (var-value)
                            (cond
                             ((not
                               (unused-env-cell var-value))
                              (let* ((var
                                      (list '?
                                            (first
                                             var-value)
                                            0
                                            0))
                                     (value
                                      (ultimate-inst
                                       var)))
                                    (cons var value)))))

                   (cxr 0 (cxr 0 environment)))))
```

```
(defun print-top-level-bindings nil
   (mapc '(lambda (var-value)
                  (cond
                   ((not (unused-env-cell var-value))
                    (let* ((var
                            (list '?
                                  (first var-value)
                                  0
                                  0))
                           (value (ultimate-inst var)))
                          (pp-external-form
                           (list var '= value))
                          (terpr)))))

         (cxr 0 (cxr 0 environment))))

(defun unique-name (term)
   (let ((unique-name-list nil)) (unique-name-1 term)))

(defun unique-name-1 (term)
   (cond ((atom term) term)
         ((variable-p term)
          (let ((old-new-name
                 (assoc term unique-name-list)))
               (cond (old-new-name (rest old-new-name))
                     (t
                      (unique-name-2 term
                                     unique-name-list)))))
         (t
          (cons (unique-name-1 (first term))
                (unique-name-1 (rest term))))))

(defun unique-name-2 (var name-list)
   (cond ((null name-list)
          (let ((subst (list '? (name-of var))))
               (setq unique-name-list
                     (cons (cons var subst)
                           unique-name-list))
               subst))
         ((equal (name-of var)
                 (name-of (first (first name-list))))
          (let ((subst (list '? (gensym))))
               (setq unique-name-list
                     (cons (cons var subst)
                           unique-name-list))
               subst))
         (t (unique-name-2 var (rest name-list)))))

(defun concat-trail (trail1 trail2)
   (cond ((atom trail1) trail2)
         (t
          (concat-trail (rest trail1)
                        (cons (first trail1) trail2)))))
```



```
(defun trace-inst (term flag)
   (cond ((equal 'exit flag) (ultimate-inst term))
         (t term)))

(defun trace-unify-p (goal db-left)
   (cond ((null db-left) nil)
         ((trace-unify-1 goal
                         (s-conclusion (first db-left))
                         '((bottom-of-env))))
         (t (trace-unify goal (rest db-left)))))

(defun trace-unify (goal db-left)
   (cond ((null db-left) 0)
         ((trace-unify-1 goal
                         (s-conclusion (first db-left))
                         '((bottom-of-env)))
          (add1 (trace-unify goal (rest db-left))))
         (t (trace-unify goal (rest db-left)))))

(defun trace-unify-1 (x y env)
   (let ((x (trace-assoc x env)) (y (trace-assoc y env)))
        (cond ((eq x y) env)
              ((or (anonym-var-p x) (anonym-var-p y)) env)
              ((variable-p x)
               (cond ((equal x y) env)
                     (t (cons (cons x y) env))))
              ((variable-p y) (cons (cons y x) env))
              ((and (numberp x) (equal x y)) env)
              ((and (atom x) (equal x y)) env)
              ((or (atom x) (atom y)) nil)
              (t
               (let ((new-env
                      (trace-unify-1 (first x)
                                     (first y)
                                     env)))
                    (and new-env
                         (trace-unify-1 (rest x)
                                        (rest y)
                                        new-env))))))))

(defun trace-assoc (x env)
   (cond ((variable-p x)
          (let ((binding (assoc x env)))
               (cond ((null binding) x)
                     (t (trace-assoc (rest binding) env)))))
         (t x)))
```

Datei : primitives.2.1

```
(defmacro is-primitive (goal)
  `(member (first ,goal) primitives))

(defmacro is-p (goal)
  `(eq (first ,goal) 'is))

(defmacro not-p (goal)
  `(eq (first ,goal) 'not))

(defmacro arith-p (goal)
  `(and (equal (length ,goal) 4)
        (assoc (first ,goal) arith-predicates)))

(defmacro s-1-ofarith (x)
  `(second ,x))

(defmacro s-2-ofarith (x)
  `(third ,x))

(defmacro s-3-ofarith (x)
  `(fourth ,x))

(defun prolog-primitive (goal)
   (cond ((arith-p goal)
          (execute-arith (first goal)
                         (s-1-ofarith goal)
                         (s-2-ofarith goal)
                         (s-3-ofarith goal)))
         ((is-p goal)
          (execute-is (second goal) (third goal)))
         ((not-p goal) (execute-not (rest goal)))
         ((eq (first goal) 'var)
          (variable-p (second goal)))
         ((eq (first goal) 'nonvar)
          (not (variable-p (second goal))))))

(defun execute-arith (fun s-1 s-2 s-3)
   (cond
    ((and (not (variable-p s-1))
          (not (variable-p s-2))
          (variable-p s-3))
     (unify s-3
            (apply (second (assoc fun arith-predicates))
                   (list s-1 s-2))
            t))))

(defun execute-not (list-of-goals)
   (not (prove (unique-name list-of-goals) 1 t)))
```



```
(defun execute-is (variable lispexpr)
   (let ((quoted-lispexpr
          (if quote-lisp-calls
              (quote-lambda-arg lispexpr)
              lispexpr)))
        (if (contains-freevars quoted-lispexpr)
            nil
            (unify variable
                   (catch-all-errors (eval quoted-lispexpr)
                                     (throw
                                      (list 'error--execute-is:eval-not-pos
                                            quoted-lispexpr)))
                   t))))

(setq primitives
      '(addit subit multit divit is not var nonvar))

(setq arith-predicates
      '((addit plus)
        (subit diff)
        (multit times)
        (divit quotient)))
```

Datei : lispeval.2.1

```
(defmacro is-lisp-defined (goal)
  `(getd (first ,goal)))

(defun lisp-predicates (goal)
   (let ((quoted-goal
          (if quote-lisp-calls
              (quote-lambda-arg goal)
              goal)))
        (cond ((contains-freevars quoted-goal) nil)
              (t
               (catch-all-errors (eval quoted-goal)
                                 (throw
                                  (list 'error--lisp-predicates:eval-not-poss:
                                         quoted-goal))))))))

(defmacro quoted-p (expr)
  `(and (consp expr) (eq (first expr) 'quote)))

(defmacro lambda-appl-p (expr)
  `(cond ((atom expr) nil)
         (t
          (or (eq (discipline (first expr)) 'lambda)
              (eq (discipline (first expr)) 'lexpr)))))

(defmacro nlambda-appl-p (expr)
  `(cond ((atom ,expr) nil)
         (t (eq (discipline (first ,expr)) 'nlambda))))

(defmacro macro-appl-p (expr)
  `(cond ((atom ,expr) nil)
         (t (eq (discipline (first ,expr)) 'macro))))

(defun application-p (expr)
   (or (lambda-appl-p expr) (nlambda-appl-p expr)))

(defun discipline (funct)
   (cond ((numberp funct) nil)
         ((atom funct)
          (let ((def (getd funct)))
               (cond ((null def) nil)
                     ((atom def) (getdisc (getd funct)))
                     (t (first def)))))
         (t (first funct))))
```



```
(defun contains-freevars (expr)
   (cond ((or (atom expr)
              (quoted-p expr)
              (nlambda-appl-p expr))
          nil)
         ((variable-p expr) t)
         ((lambda-appl-p expr)
          (some (function contains-freevars) (rest expr)))))

(defun quote-lambda-arg (expr)
   (let ((expr (expand-macro expr)))
        (cond ((or (null expr) (numberp expr)) expr)
              ((application-p expr)
               (quote-application expr))
              (t (list 'quote expr)))))

(defun quote-nlambda-arg (expr)
   (let ((expr (expand-macro expr)))
        (cond ((atom expr) expr)
              ((application-p expr)
               (quote-application expr))
              (t
               (mapcardot (function quote-nlambda-arg)
                          expr))))))

(defun quote-application (expr)
   (cond ((or (variable-p expr) (quoted-p expr)) expr)
         ((lambda-appl-p expr)
          (cons (first expr)
                (mapcar (function quote-lambda-arg)
                        (rest expr))))
         ((nlambda-appl-p expr)
          (cons (first expr)
                (mapcar (function quote-nlambda-arg)
                        (rest expr)))))))

(defun expand-macro (expr)
   (cond ((macro-appl-p expr) (macroexpand expr)) (t expr)))
```

Datei : interface.2.1

```
(setq lisp-coms
      '((ass ass
             "Einfuegen einer Klausel in die Datenbasis"
             ass.help)
        (+ ass "Kurz fuer ass" ass.help)
        (rex rex "Loeschen einer Klausel" rex.help)
        (- rex "Kurz fuer rex" rex.help)
        (abolish abolish
                 "Loeschen eines Praedikates"
                 abolish.help)
        (destroy destroy
                 "Loeschen aller Praedikate"
                 abolish.help)
        (sort-db sort-db "Sortieren der Datenbasis" nil)
        (listing listing
                 "Auflisten der angegebenen Praedikate, nil = alles"
                 listing.help)
        (l listing "Kurz fuer listing" listing.help)
        (lisp do-exit-lisplog
              "Verlasse den LISPLOG-Interpreter"
              nil)
        (edit edit "Editieren des angegebenen Praedikates" nil)
        (tell tell "Speichern der Dateibasis" save.help)
        (consult consult "Laden der angegebenen Datei" save.help)
        (spy spy "Tracen der angegebenen Praedikate" spy.help)
        (nospy nospy
               "Tracer fuer angegebene Praedikate ausschalten"
               spy.help)
        (brk brk "Break fuer angegebene Praedikate" brk.help)
        (nobrk nobrk "Break ausschalten" brk.help)
        (cut mcut "Hand-Cut fuer angegebene Praedikate" cut.help)
        (nocut nomcut "Handschneider ausschalten" cut.help)
        (auto-quote-on auto-quote-on
                       "Quotierungsautomatik anschalten"
                       nil)
        (auto-quote-off auto-quote-off
                        "Quotierungsautomatik ausschalten"
                        nil)
        (help do-help "Drucke Hilfsinformationen" nil)))

(setq lisplog-top-level-prompt '*)

(defun reset-ctrl-c (x)
   (newintr)
   x)

(defun set-ctrl-c (x)
   (cond (tracemode (signal 2 'ctrl-c)))
   x)
```



```
(defun commands (eingabe)
   (let* ((inline (if (atom eingabe) (list eingabe) eingabe))
          (command (assoc (first inline) lisp-coms))
          (anfrage
           (if (atom (first inline)) (list inline) inline)))
         (cond (command
                (let ((toplevel-flag t))
                     (eval
                      (cons (second command)
                            (rest inline)))))
               (t (cond (tracemode (tracer-init) (terpr)))
                  (cond ((not (atom tracemode)) (terpr)))
                  (set-ctrl-c nil)
                  (reset-ctrl-c
                   (catch (prove anfrage nil nil)))))))

(defun y-or-n-p (message)
   (patom message)
   (let ((response (read)))
        (cond ((eq response 'y) t)
              ((eq response 'n) nil)
              ((status isatty) (y-or-n-p message)))))

(defun lisplog nil
   (let ((toplevel-flag nil))
        (do ((leave-prolog nil))
            (leave-prolog)
            (patom lisplog-top-level-prompt)
            (pp-external-form
             (catch-all-errors (commands (linetest nil))
                               "error--lisp:internal ???"))
            (terpr))))

(defun n-solutions (goal anzahl)
   (prove (list goal) anzahl nil))

(defun do-exit-lisplog nil
   (setq leave-prolog t))

(defun auto-quote-on nil
   (setq quote-lisp-calls t))

(defun auto-quote-off nil
   (setq quote-lisp-calls nil))
```

```
(defun do-external-help (argument)
   (let ((command (assoc (first argument) lisp-coms)))
        (cond ((null command)
               (apply (function help) argument))
              ((fourth command)
               (*process
                (implode
                 (append (append (explode
                                  '/usr/users/lisplog/tools/LISPLOG-help)
                                 '(| |))
                         (explode (fourth command))))))
              (t (patom "Keine weitere Hilfe verfuegbar")
                 (terpri)))))

(def do-help
  (nlambda (argument)
    (cond ((null argument) (print-help))
          (t (do-external-help argument)))))

(defun print-help nil
   (patom "Folgende Kommandos stehen zur Verfuegung:")
   (terpri)
   (mapc '(lambda (command)
                 (print (first command))
                 (n-spaces
                  (- 15 (length (explode (first command)))))
                 (patom " : ")
                 (patom (third command))
                 (terpri))

          lisp-coms)
   t)
```



```
(defun lisplog-start-message nil
   (patom "LISPLOG System geladen.")
   (terpr)
   (terpr)
   (patom "Diese Version umfasst :")
   (terpr)
   (patom "   - LISPLOG Interpreter Version 2")
   (terpr)
   (patom "   - Box Modell Tracer")
   (terpr)
   (patom "   - Break Paket")
   (terpr)
   (patom "   - schaltbare Quotierungsautomatik")
   (terpr)
   (terpr)
   (patom "starte LISPLOG mit  (lisplog)")
   (terpr)
   (patom "weitere Informationen dann mit   help")
   (terpr)
   (patom "   ")
   (terpr)
   (patom
    "Auf dem LISPLOG-Toplevel habe alle Kommandos die Form : ")
   (terpr)
   (patom "   <Kommando> {<argumente>}*")
   (terpr)
   (patom "   ")
   (terpr)
   (patom "Namen, die keine Kommandos sind,")
   (terpr)
   (patom
    "werden als Anfragen eines LISPLOG-Goals interpretiert.")
   (terpr)
   (terpr)
   (patom
    "Die Form (<g1> {<a1>}*) (<g2> {<a2}*) ...   kann benutzt werden,")
   (terpr)
   (patom
    "um mehrere konjunktiv verknuepfte Ziele zu beweisen.")
   (terpr))
```

Datei : extern.2.1

```
(defmacro set-underscore-syntax
          nil
          '(setsyntax '_
                      'vmacro
                      (function
                       (lambda nil
                               (list '? (read)))
                       )))

(defmacro reset-underscore-syntax
          nil
          '(setsyntax '_ 2))

(set-underscore-syntax)

(putprop 'cut '! 'printmacrochar)

(setsyntax '!
           'vmacro
           (function (lambda nil
                          (list 'cut (read)))
                     ))

(defun external-form (term)
   (cond ((atom term) term)
         ((variable-p term)
          (cond ((or (null (level-of term))
                     (zerop (level-of term)))
                 (uconcat '"_" (name-of term)))
                (t
                 (uconcat '"_"
                          (name-of term)
                          '"_"
                          (level-of term)))))
         (t
          (cons (external-form (first term))
                (external-form (rest term))))))

(defun pp-external-form (expr &rest port)
   (let ((external-expr (external-form expr)))
        (reset-underscore-syntax)
        (pp-form external-expr
                 (cond (port (car port)) (t nil)))
        (set-underscore-syntax)))
```



```
(defun print-external (expr &rest port)
   (let ((external-expr (external-form expr))
         (prinlength 4)
         (prinlevel 4))
        (reset-underscore-syntax)
        (print external-expr
               (cond (port (car port)) (t nil)))
        (set-underscore-syntax)))
```

Anhang 2 : Laufzeitanalyse von LISP-Programmen

Dieses Tool ermoeglicht es, die Verteilung der Programmlaufzeit auf ausgewaehlte Funktionen zu bestimmen. Man kann jede beliebige LISP-Funktion ueberwachen. Fuer eine ueberwachte Funktion wird die Zeit gemessen, die zur Ausfuehrung dieser Funktion benoetigt wird. Durch die Ueberwachung laufen die Programme erheblich langsamer, gemessen werden also nicht absolute Groessen, sondern die Relation zwischen den ueberwachten Funktionen.

Ueberwacht man die Funktionen 'a' und 'b', waehrend man 'c' nicht ueberwacht, und ruft 'a' die Funktionen 'b' und 'c', so wird die benoetigte Zeit fuer 'a' und 'c' unter 'a' abgerechnet, waehrend die Zeit fuer 'b' unter 'b' abgerechnet wird und in der Summe fuer 'a' nicht enthalten ist. Diese Abrechnung arbeitet bei beliebigen Aufruffolgen, also auch bei korekursiven Funktionen. Auch compilierte Funktionen koennen abgerechnet werden, wenn sie nicht lokal definiert sind.

Zur Steuerung der Ueberwachung stehen dem Benutzer die folgenden Funktionen zur Verfuegung :

(control-fun (f1) (f2) ...)	die Funktionen (f1) (f2) ... werden ueberwacht
(no-control-fun)	keine Funktion wird ueberwacht
(no-control-fun (f1) (f2) ...)	die Funktionen (f1) (f2) ... werden nicht mehr ueberwacht
(time-control-on)	Begin der Ueberwachung (Zaehler Reset)
(time-control-off)	Ende der Ueberwachung
(control-print)	Ausdruck des Ergebnisses

In den ausgegebenen Zeiten ist der Aufwand fuer event. Gargabe Collection nicht (!) enthalten, weil dadurch das Ergebnis verfaelscht wuerde. Die Zeit fuer Garbage Collection wird seperat gemessen. Dieses Programm benutzt den LISP-Tracer; Funktionen, die zeitueberwacht werden, koennen nicht gleichzeitig getraced werden.

Implementierung

Property	'time-spent	Value : Number
Global Var.	control-fun-list	Value : List of Symbol
Global Var.	running-fun	Value : List of Symbol
Global Var.	last-switch	Value : Number
Global Var.	total-time	Value : Number
Global Var.	total-gc	Value : Number


```
(declare (macros t)
         (special control-fun-list
                  running-fun
                  last-switch
                  total-time
                  total-gc)
         (nlambda control-fun
                  no-control-fun)
         )

(setq control-fun-list '(*top-level*))

(defun time-control-on nil
   (mapc '(lambda (arg)
                   (putprop arg 0 'time-spent))

         control-fun-list)
   (setq running-fun '(*top-level*))
   (setq last-switch (current-time))
   (setq total-time last-switch)
   (setq total-gc (cadr (ptime)))
   '
   time-control-started)

(defun time-control-off nil
   (setq total-time (diff (current-time) total-time))
   (setq total-gc (diff (cadr (ptime)) total-gc))
   '
   time-control-ended)

(def control-fun
  (nlambda (fun)
    (mapc '(lambda (arg)
                   (apply (function trace)
                          (list
                           (list arg
                                 'traceenter
                                 'switch-to
                                 'traceexit
                                 'switch-back)))
                   (putprop arg 0 'time-spent))

           fun)
    (setq control-fun-list (append fun control-fun-list))))
```

```
(def no-control-fun
  (nlambda (fun)
    (let ((fun (cond ((null fun) control-fun-list) (t fun))))
         (mapc '(lambda (arg)
                        (apply (function untrace) (list arg))
                        (remprop arg 'time-spent))
               fun)
         (setq control-fun-list
               (remove-list fun control-fun-list)))
    (cond
      ((not (member '*top-level* control-fun-list))
       (setq control-fun-list
             (cons '*top-level* control-fun-list))))))

(defun control-print nil
   (patom "Total Time is : ")
   (print total-time)
   (terpr)
   (patom "Total GC is : ")
   (print total-gc)
   (terpr)
   (mapc '(lambda (arg)
                  (print (get arg 'time-spent))
                  (patom " (")
                  (print
                   (quotient (times 100
                                    (get arg
                                         'time-spent))
                             total-time))
                  (patom "`%) spent for ")
                  (print arg)
                  (terpr))
         control-fun-list))

(defun remove-list (remlex lex)
   (cond ((null lex) nil)
         ((member (car lex) remlex)
          (remove-list remlex (cdr lex)))
         (t (cons (car lex) (remove-list remlex (cdr lex))))))

(defun switch-to (arg dummy)
   (let ((now (current-time)))
        (putprop (car running-fun)
                 (add (get (car running-fun)
                           'time-spent)
                      (diff now last-switch))
                 'time-spent)
        (setq running-fun (cons arg running-fun))
        (setq last-switch now)))
```



```
(defun switch-back (arg dummy)
   (cond ((equal arg (car running-fun))
          (let ((now (current-time)))
               (putprop (car running-fun)
                        (add (get (car running-fun)
                                  'time-spent)
                             (diff now last-switch))
                        'time-spent)
               (setq running-fun (cdr running-fun))
               (setq last-switch now)))
         (t (print "Error in time controll") (terpr))))

(defun current-time nil
   (diff (car (ptime)) (cadr (ptime))))
```

Anhang 3 : Speicherverbrauchsanalyse fuer LISP-Programme

Dieses Tool ermoeglicht es, die Verteilung des Speicherverbrauchs auf ausgewaehlte Funktionen zu bestimmen. Man kann jede beliebige LISP-Funktion ueberwachen. Fuer eine ueberwachte Funktion wird der Speicherverbrauch waehrend der Ausfuehrung dieser Funktion gemessen. Durch die Ueberwachung laufen die Programme erheblich langsamer.

Speicherverbrauch heisst hier die Verwendung von Cons-Zellen und anderen Speicherbereichen. Um den gesamten Speicherverbrauch zu erfassen, muss man in diesem Ueberwachungsprogramm die Basisfunktionen des FRANZ-LISP Systems angeben, die Speicher verbrauchen. Zur Zeit sind folgenden Funktionen bruecksichtigt :

Funktion	Speicherverbrauch
(append a b)	(length a)
(cons a b)	1
(list a b c ..)	(length a b c ..)
(reverse a)	(length a)
(mapcar f a)	(length a)
(ncons a)	1
(makhunk x)	x

Benutzt man noch andere Basisfunktionen, so muessen diese ins Programm aufgenommen werden (vgl. Listing). Werden sie nicht aufgenommen, so wird der durch sie verbrauchte Speicher nicht erfasst (Messfehler).

Zur Steuerung der Ueberwachung stehen dem Benutzer die folgenden Funktionen zur Verfuegung :

(control-fun <f1> <f2> ...)	die Funktionen <f1> <f2> ... werden ueberwacht
(no-control-fun)	keine Funktion wird ueberwacht
(no-control-fun <f1> <f2> ...)	die Funktionen <f1> <f2> ... werden nicht mehr ueberwacht
(mem-control-on)	Begin der Ueberwachung (Zaehler Reset)
(mem-control-off)	Ende der Ueberwachung
(control-print)	Ausdruck des Ergebnisses

Implementierung

Property	'memory-used	Value : Number
Global Var.	control-fun-list	Value : List of Symbol
Global Var.	running-fun	Value : List of Symbol

Beachte : Die Datei kann nicht compiliert werden !


```
(putd 'cons-nt (getd 'cons))

(putd 'append-nt (getd 'append))

(putd 'list-nt (getd 'list))

(putd 'mapcar-nt (getd 'mapcar))

(declare (macros t)
         (special control-fun-list running-fun)
         (nlambda control-fun
                  no-control-fun)
         )

(setq control-fun-list '(*top-level*))

(setq running-fun '(*top-level*))

(defun mem-control-on nil
   (mapc '(lambda (arg)
                  (putprop arg 0 'memory-used))

         control-fun-list)
   (setq running-fun '(*top-level*))
   (trace-memory-user)
   '
   mem-control-started)

(defun mem-control-off nil
   (apply (function untrace)
          '(append cons list reverse mapcar ncons makhunk))
   '
   mem-control-ended)

(def control-fun
  (nlambda (fun)
    (mapc '(lambda (arg)
                   (apply (function trace)
                          (list-nt
                           (list-nt arg
                                    'traceenter
                                    'switch-to
                                    'traceexit
                                    'switch-back)))
                   (putprop arg 0 'memory-used))

          fun)
    (setq control-fun-list (append-nt fun control-fun-list))))
```

```
(def no-control-fun
  (nlambda (fun)
    (let ((fun (cond ((null fun) control-fun-list) (t fun))))
         (mapc '(lambda (arg)
                          (apply (function untrace) (list-nt arg))
                          (remprop arg 'memory-used))

               fun)
         (setq control-fun-list
               (remove-list fun control-fun-list)))
    (cond
     ((not (member '*top-level* control-fun-list))
      (setq control-fun-list
            (cons-nt '*top-level* control-fun-list))))))

(defun control-print nil
   (patom "Total Memory used : ")
   (let ((total-mem
          (apply (function add)
                 (mapcar-nt '(lambda (arg)
                                     (get arg
                                          'memory-used))

                            control-fun-list))))
        (print total-mem)
        (terpr)
        (mapc '(lambda (arg)
                         (print (get arg 'memory-used))
                         (patom " (")
                         (print
                          (quotient (times 100.0
                                           (get arg
                                                'memory-used))
                                    total-mem))
                         (patom " %) spent for ")
                         (print arg)
                         (terpr))

              control-fun-list)))

(defun remove-list (remlex lex)
   (cond ((null lex) nil)
         ((member (car lex) remlex)
          (remove-list remlex (cdr lex)))
         (t
          (cons-nt (car lex)
                   (remove-list remlex (cdr lex))))))

(defun switch-to (arg dummy)
   (setq running-fun (cons-nt arg running-fun)))
```



```
(defun switch-back (arg dummy)
   (cond ((equal arg (car running-fun))
          (setq running-fun (cdr running-fun)))
         (t (print "Error in memory controll") (terpr))))

(defun add-to-current (n)
   (putprop (car running-fun)
            (add n
                 (get (car running-fun) 'memory-used))
            'memory-used))

(defun dummy (a b)
   nil)

(defun enter-cons (fun args)
   (add-to-current 1))

(defun enter-append (fun args)
   (add-to-current (length (car args))))

(defun enter-list (fun args)
   (add-to-current (length args)))

(defun enter-reverse (fun args)
   (add-to-current (length (car args))))

(defun enter-mapcar (fun args)
   (add-to-current (length (cadr args))))

(defun enter-ncons (fun args)
   (add-to-current 1))

(defun enter-makhunk (fun args)
   (add-to-current (car args)))

(defun trace-memory-user nil
   (apply (function trace)
          '((cons traceenter enter-cons traceexit dummy)
            (append traceenter enter-append traceexit dummy)
            (list traceenter enter-list traceexit dummy)
            (reverse traceenter
                     enter-reverse
                     traceexit
                     dummy)
            (mapcar traceenter enter-mapcar traceexit dummy)
            (ncons traceenter enter-ncons traceexit dummy)
            (makhunk traceenter
                     enter-makhunk
                     traceexit
                     dummy))))
```